1 Introduction

Any component model suitable for hard-real time systems, \(1\) must be able to specify and monitor real time properties. CCM does not support specification for task periodicity and deadline. A hard real-time component model, must allow certain component tasks to be defined as periodic. Moreover, all tasks executed by a component must specify deadlines. In addition, these deadlines must be monitored for violation. \(2\) Moreover, in order to be able to analyze the system at design time we must be able to ascertain the semantics of component interaction i.e. the model must have well-defined interaction semantics.

In addition, there are some additional guidelines, which must be followed. Given that typical hard real time systems require determinism, components must support static memory allocation i.e. the resources required for execution of tasks must be created at system initialization. Another advantage of this approach is that it avoids memory fragmentation, which if present requires the RTOS to spend precious computing cycles managing the memory fragmentation. The consequence of static memory allocation is that components can only support service life cycle i.e. a session-based component cannot be created dynamically based on an incoming request. This is especially true in hard real time systems, where all possible tasks must be known before hand to ensure that they can be scheduled.

Another feature, which is not a requirement for hard-real time systems but is a good feature to have is spatial and temporal partitioning. However, a safety-critical real-time system, like ARINC-653 \[1\] compliant operating systems, can group processing modules into partitions with different criticality levels and still allow them to execute in the same core module, without affecting one another spatially or temporally. This is a good feature to have in a software assembly, where components with different criticality levels can be isolated from each other, which in turn provides fault isolation.

**Remark 1** We have named our component model ARINC-653 Component Model (ACM) because ARINC-653 standard is the current state of the art for safety critical real-time systems that supports the two platform principles mentioned in previous paragraphs. However, we should clarify that this component model is not necessarily tied to ARINC-653 and can be implemented on any other platform that provides similar services.

Figure 1 illustrates the internals of our component model. A component can have four different kinds of ports - consumer port, publisher port, provided interface port (similar to a facet in CCM) and required interface port (similar to a CCM receptacle). A publisher port is a source of events: this port is used to produce events that will be consumed by another component/s. A publisher port needs to be triggered to publish an event (probably read from some internal state variable or a hardware source). This triggering can be either periodic or aperiodic (sporadic). While, a periodic publisher is triggered at regular intervals by a clock to supply data, an aperiodic publisher is invoked (sporadically) by an internal method, possibly the implementation code of another port. A consumer port as the name suggests acts as a sink for events. Like a publisher port, it can be triggered periodically (by a clock) or aperiodically (by the arrival of an event) to consume an event. While an aperiodic consumer consumes all the events published by its publisher on a FIFO basis (destructive read), a periodic consumer samples the events published at a specified rate (nondestructive read).

A provided interface port or facet contains the implementation for the methods defined in the provided interface and services the request issued on these interfaces by a receptacle. The incoming client requests are queued by the middleware and are serviced by the provided port’s implementation in the FIFO order. Two new concepts exist in our extension to the CCM: state variables, which are similar to attributes in CCM but cannot be modified from outside.
component, and component triggers, which are internal periodically activated methods within a component that can be used for internal bookkeeping and checking state invariants.

All the ports - publisher, consumer, facet, and receptacle - and the Component Trigger method have to finish their unit of work within a specified deadline. This deadline can be qualified as HARD (strict) or SOFT (relatively lenient). A HARD deadline violation is an error that requires intervention from the underlying middleware. A SOFT deadline violation results in a warning.

Like the deadline, all implementations can specify another property that must be respected - contracts. These contracts are expressed as pre-conditions and/or post-conditions. Any contract violation results in an error. This concept is based upon the logic system identified by Hoare [5]. The key feature of this logic are assertions of form \( \{ \text{pre} \} P \{ \text{post} \} \) commonly known as Hoare Triple, where \( P \) is a computer program. These are discussed further in section 3.
Figure 2: ACM Metamodel. The colored boxes indicate the four types of ports, which can be used to interact with other components.

Next section describes the metamodel i.e. the relationship and attributes associated with these ports.

2 ACM Metamodel

Figure 2 describes the metamodel of our extensions to the CORBA Component Model. This metamodel has been designed in a UML based modeling language called MetaGME [6, 2]. It shows the relationship between all the ports and different attributes that can be set. The main concepts are:

(a) **State Variables**: As described in the previous section, these represent the hidden and externally observable state of the component. Unlike CCM attributes they cannot be modified from outside.

(b) **Parameters** are configuration attributes which once set during initialization remain constant during the life cycle of that component.

(c) **Asynchronous Ports**: These kinds of ports participate in asynchronous interactions. There are two kinds of asynchronous ports, **Publisher** and **Consumer**. We discussed both of them in the previous section. The data type of an event is specified by the type entity referred to by the port. Both ports can be further qualified by attributes such as period, deadline, pre-conditions and post-conditions. (see Table 1). If the period of a publisher is set to infinity, it is called aperiodic. Otherwise, it is periodic. Periodic publishers are typically activated by the middleware, automatically, while aperiodic publishers are activated by another method within the component. Periodic consumers are automatically and periodically activated by the middleware, aperiodic Consumers are activated when they have events to consume.

(d) **Synchronous Ports**: These ports are used for either requesting some service from another component or servicing an incoming request. There are two main kinds of synchronous ports, **Provided** and **Required**. Each provided port contains implementation of a collection of **Methods**. These methods can be further qualified by filling in their attributes e.g. call type, deadline, pre conditions and post conditions. (see Table 1). Furthermore, each method contains a number of arguments. These arguments can be marked to be used strictly for sending input (IN) to the facet, strictly for sending output (OUT) to the receptacle, or to be used for both input as well as output (INOUT). The third
kind. **Resource Usage Monitoring Interface** is used for monitoring the component resource usage. By design, this interface can only contain read-only methods i.e. they cannot change the internal state of the component.

**(e) Component Triggers:** A component can contain a number of internal methods, known as component triggers. These methods are periodically triggered. They must have a finite, non-zero period. Typically, they are used for record keeping and invariant checking of the component. These methods are further qualified by filling in attributes such as period, deadline and invariant (see Table 1).

**(f) Component Health Managers:** Component-level health managers (CLHM) for software components detect anomalies, identify and isolate the fault causes of those anomalies (if feasible), prognosticate future faults, and mitigate effects of faults – on the level of individual components. We envision CLHM implemented as a ‘side-by-side’ object that is attached to a specific component and acts as its health manager. It provides a localized and limited functionality for managing the health of one component, but it also reports to higher-level health manager(s) (the system health manager) - all in a real-time context where dependability is required. A detailed discussion on this topic is out of scope for this paper. Interested readers can refer to [4, 3].

Unlike the standard CCM where the functional logic belonging to an interface port is executed on a new, dynamically created, or pre-existing but dynamically released worker-thread, in ACM the functional logic for each port is executed on a statically allocated schedulable unit. This choice is guided by our first design principle of static memory allocation, which restricts dynamic creation. On an ARINC-653 operating system, these schedulable units are ARINC-653 processes.

### 3 Correctness Contracts

Each functional entity in ACM can be annotated with correctness criteria, which are specified as either pre-conditions or post conditions. We envision that these conditions should be specified over the current value or the historical change in the value, or rate of change of values of certain parameters. These parameters can be (a) the event-data of asynchronous calls, or (b) function-parameters of synchronous calls, or (c) state variables of the component, or (d) resource usage of the component. While pre-conditions are assumptions that must be true before execution of a call, post conditions are guarantees, which must be correct. This type of reasoning is critical in achieving modular certification of software components [7].

Next section describes the semantics of component interactions.

### 4 Semantics of Component Interactions

While each component and its associated ports, states, internal triggers can be individually configured, an assembly is not complete until the interactions between the component ports is configured. The association between the ports depends on their type (synchronous/asynchronous) and the event/interface type associated with the port. Two kinds of interactions, asynchronous interactions and synchronous interactions are possible between components. The possible combination of these interactions with periodic and aperiodic triggering of processes that are bound to the respective ports gives rise to a richer set of behaviors compared to CCM.

#### 4.1 Asynchronous Interactions

These interactions occur when a publish port of a component is connected to a consumer port of another component. While a consumer can be connected to only one publisher, a publisher may be connected to one or more consumers. Strict type matching on the event type is required between the publisher and its consumers.

A periodic consumer always exhibits sampling behavior. Even if the rate of the publisher is indeterminate, for example if the publisher is aperiodic, setting the period of the consumer ensures that the events from the publisher are sampled at a specific rate. When the interacting publisher and consumer both are periodic, the value of the consumer’s period relative to the publisher’s determines if the consumer is over-sampling (higher rate of consumption or lower period compared to publisher) or under-sampling (lower rate of consumption or higher periodicity compared to publisher).

Interaction between a periodic publisher and an aperiodic consumer is indicative of a pattern where the sink or the consumer is reactive in nature. In such a case, the consumer port stores incoming published events in a queue, which
Table 1: Attributes of Component Entities

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Values</th>
<th>Default</th>
<th>Remark</th>
<th>Applicable To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validity Period</td>
<td>double</td>
<td>$(0, \infty)$</td>
<td>$\infty$</td>
<td>Marks the time-limit for an event to be valid (not stale).</td>
<td>Consumers</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>Boolean</td>
<td>True, False</td>
<td>$\infty$</td>
<td>Marks if the port can update the state of the component or not</td>
<td>All ports and component triggers</td>
</tr>
<tr>
<td>Deadline</td>
<td>double</td>
<td>$(0, \infty)$</td>
<td>$\infty$</td>
<td>Marks the time-limit by which the process should finish execution.</td>
<td>All ports and component triggers</td>
</tr>
<tr>
<td>Deadline Type</td>
<td>ENUM</td>
<td>HARD</td>
<td>HARD</td>
<td>Marks the nature of the deadline.</td>
<td>All ports and component triggers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SOFT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Period</td>
<td>double</td>
<td>[Deadline, $\infty$]</td>
<td>$\infty$</td>
<td>Marks the rate at which the scheduler launches the process.</td>
<td>All ports and component triggers</td>
</tr>
<tr>
<td>Call Type</td>
<td>ENUM</td>
<td>TWO WAY</td>
<td>TWO WAY</td>
<td>Marks whether an RMI call is blocking or non-blocking (same as in CCM/CORBA)</td>
<td>Methods on Synchronous ports</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ONE WAY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostCondition</td>
<td>string</td>
<td>N/A</td>
<td>N/A</td>
<td>The contract to be satisfied at the end of the execution.</td>
<td>All ports</td>
</tr>
<tr>
<td>PreCondition</td>
<td>string</td>
<td>N/A</td>
<td>N/A</td>
<td>The contract to be satisfied at the beginning of the execution.</td>
<td>All ports</td>
</tr>
<tr>
<td>Invariant</td>
<td>string</td>
<td></td>
<td></td>
<td>The contract to be satisfied during the operation of trigger.</td>
<td>Component triggers</td>
</tr>
</tbody>
</table>

are consumed in a FIFO manner. If the queue size is configured appropriately, this allows the consumer to operate on all of the events received.

The case for interaction between an aperiodic publisher and an aperiodic consumer is similar to the one between a periodic publisher and an aperiodic consumer.

4.2 Synchronous Interactions

This interaction implies call-return semantics. A required interface port can be associated with a provided interface port of an identical interface type. A provides port can be associated with one or more requires ports. Because of the synchronous nature of these interactions, the deadline of required interface method (i.e. the caller) must be greater than the deadline value for the provided interface method (i.e. the callee).

Synchronous ports in this model are always aperiodic. The interaction patterns observed in synchronous ports is borrowed from CCM. The key difference is deadline monitoring. The default type of interaction is call-return or two-way communication i.e. the Requires port waits for the Provides port to finish its operation and return the results.

Recently, we relaxed the restrictions on synchronous interactions to allow CORBA style one-way calls. When such methods are invoked, the Requires port performs a non-blocking call. It returns without waiting for the Provides port to finish its operation. There are no return values in such calls. However, one should note that even though the call is made in a non-blocking fashion it is different from an asynchronous interaction. While, a publisher does not fail if a consumer fails to consume the message properly, a one-way call via the middleware will result in an exception if the target provided port is not available.

**Remark 2** Safe component interactions must satisfy the following constraints: (a) Deadline of a requires port must be greater than or equal to the deadline of the interacting provided port. (b) Validity period of a consumer must be greater than the periodicity of the publisher. Otherwise, a consumer can possibly receive data that it considers stale. (c) The contract imposed by the post condition of a component providing a service or publishing an event must be stricter than the precondition checked by the interacting destination component. If this is not true, the source component might send data, which is locally valid but will violate precondition of the destination component.
Remark 3 It is important to point out that it is possible to identify specific faults and their propagation pattern based on component interactions - periodic publisher/periodic consumer, periodic publisher/aperiodic consumer aperiodic publisher/periodic consumer, aperiodic publisher/aperiodic consumer, and synchronous interactions. While the interaction ports can be customized (by the event-data-types published/consumed, interfaces/methods exposed, periodicity, deadline etc.), their fundamental behaviors and interaction patterns are well defined. Additionally, capturing the data and control flow model about the component internals (between the component ports), further assists in capturing the fault propagation within the component. This approach is similar to the failure propagation and transformation calculus described by Wallace [8]

5 Example

Figure 3 shows a simple example assembly of components. The Sensor component contains an asynchronous publisher interface (source port) that is triggered periodically (every 4 msec). The event published by this interface is consumed by a periodically triggered asynchronous consumer/event sink port on the GPS component (every 4 msec). Note that the event sink process is periodically released, and each such invocation reads the last event published by the Sensor. If the Sensor does not update the event frequently enough, the GPS may read stale data. The consumer process in the GPS, in turn, produces an event that is published through the GPS’s event publisher port. This event triggers the aperiodic consumer / event sink port on the Navigation Display component. Upon activation, the display component uses an interface provided by the GPS to retrieve the position data via a synchronous method invocation call into the GPS component.

References